Software Development Strategies For Streaming SIMD Extensions

Version 2.1

01/99

Order Number: 243648-002

Table of Contents

Revision History

Revision	Revision History	Date
2.1	FCS revision.	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions Using the Newton-Raphson Method*, Intel Application Note, AP-803, Order No. 243637-001

1 Introduction

Streaming SIMD Extensions for the Intel® Architecture (IA) instruction set provide single precision floating-point single-instruction, multiple-data (SIMD) instructions. These instructions provide a means to accelerate applications that rely heavily on floating-point operations, such as 3D geometry, video processing, and spatial (3D) audio. This application note addresses issues related to engineering tradeoffs of code design time versus performance, and includes examples of code that exploit the Streaming SIMD Extensions.

A series of code examples are presented and analyzed with respect to performance considerations and design time. The intent is to present the reader with examples of coding effort for the different coding methodologies. The goal is that this information can be used to determine the appropriate effort to apply when optimizing code.

2 Software Development Strategies For Streaming SIMD Extensions

Recent trends in processor architectures have introduced SIMD to mainstream computing and mainstream programming languages. SIMD has primarily migrated from the arena of scientific and supercomputing where FORTRAN is the predominant programming language. One of the advantages of FORTRAN is that it allows for relatively simple vectorization of code. Vectorization is the process of transforming sequentially executing, or scalar, code into code that can execute in parallel, taking advantage of the inherent SIMD capabilities of an architecture. Vectorizing compilers are compilers that understand the SIMD characteristics of the target architecture, and can do the transformation automatically. FORTRAN compilers in the supercomputing arena have had this capability for two decades.

As SIMD migrates into the mainstream arena, however, the language of choice is C/C++. While there are obscure methods in C/C++ for indicating parallelism in algorithms, there are few standardized approaches. Although vectorizing C/C++ compilers have appeared and are beginning to mature, much development work is still needed. As a result, much of the code that exploits SIMD capabilities to date has been hand written in assembly language (for example, code for MMX™ technology). Another reason for writing the code in assembly is that the capabilities are only utilized in performance critical algorithms and are, therefore, hand coded to realize the highest performance possible.

This application note contains different implementations of a 3D transformation algorithm using intrinsics, assembly, and a vector class implementation to show the options for writing SIMD code. The case study presented, essentially a matrix * vector + vector operation, also shows the effort required, and the return on investment for such implementations.

2.1 Description of Test Algorithm

The algorithm used in the study was a basic 3D transformation that included perspective projection and viewport scaling and translation. The viewport scaling was assumed to be incorporated into the matrix. Further details and explanations can be found in any basic computer graphics text. The specific C/C++ reference implementation is listed in Example 1. (The code for this application note also can be found in the samples\Transform\Transform.cpp file.).

```
void Transform_scalar_c(InVerts &InputVerts, OutVerts &OutputVerts, int Counter)
{
    float curX, curY, curZ, curW;
    float mat00=1.0, mat01=0.0, mat02=0.0, mat03=0.0;
    float mat10=0.0, mat11=1.0, mat12=0.0, mat13=0.0;
    float mat20=0.0, mat21=0.0, mat22=1.0, mat23=0.0;
    float mat30=0.0, mat31=0.0, mat32=0.0, mat33=1.0;
    float VX_X=1.0, VX_Y=1.0, VX_Z=1.0;
    float Xout, Yout, Zout, Wout;
    int i;

    for(i=0;i<Counter;i++)
    {
        curX    = InputVerts.x[i];
        curY    = InputVerts.y[i];
        curZ    = InputVerts.z[i];

        curW    = (float)(1.0/((curX * mat30) + (curY * mat31) + (curZ * mat32) + mat33));

        Xout    =((((curX * mat00) + (curY * mat01) + (curZ * mat02) + mat03) * curW) + VX_X);
        Yout    =((((curX * mat10) + (curY * mat11) + (curZ * mat12) + mat13) * curW) + VX_Y);
        Zout    =((((curX * mat20) + (curY * mat21) + (curZ * mat22) + mat23) * curW) + VX_Z);

        OutputVerts.x[i] = Xout;
        OutputVerts.y[i] = Yout;
        OutputVerts.z[i] = Zout;
        OutputVerts.w[i] = curW;
    }
}
```

Example 1: Reference C/C++ Implementation

Once again, this is essentially a matrix times a vector plus a vector. This algorithm was chosen for a number of reasons:

1. The expectation that it would have significant relevance to most programmers that want to utilize the capabilities of Streaming SIMD Extensions.
2. The variety of different approaches that can be used to exploit parallelism for this particular algorithm.
3. The algorithm represents a significant but not overwhelming amount of code at many levels of abstraction. This is important because of the many ways in which the new Streaming SIMD Extensions development environment provided by Intel allows for SIMD floating-point instructions.

2.2 Exploitable Parallelism

It is important to understand what types of parallelism an algorithm possesses so that an optimal implementation can be developed. It is also important to understand that the availability of exploitable parallelism can significantly affect the implementation and maintenance effort required for a piece of

code. This section describes three situations in which parallelism can be exploited, and the benefits associated with each situation.

2.2.1 Compressed Computation

Compressed computation can be obtained by finding similar operations, multiplies for example, in a piece of code that are independent of each other, and executing them at the same time by exploiting SIMD capabilities. For instance, in Example 1 the calculations required that curW include three independent multiplies. These multiplies could be executed simultaneously by loading the vector into one Pentium® III register, `xmm0`, and the fourth row of the matrix into another, `xmm1`, then issuing a "`mulps xmm0, xmm1`" instruction.

While this appears to be beneficial on the surface, it is unlikely to produce significant benefit. The issue is that once the data has been multiplied it needs to be summed. This requires the ability to sum all of the elements within a single register. However, this ability is not included in the Streaming SIMD Extensions. Consequently, a significant amount of overhead code is required to sum the partial results, which leads to less than optimal code. It is important to note that for some algorithms, compressed computation is the only type of exploitable parallelism. In such cases, the programmer must investigate the performance tradeoffs of using SIMD instructions to make sure that the SIMD code is in fact more efficient than a scalar implementation.

2.2.2 Decreased Latency

A close relative of compressed computation is decreased latency. Decreased latency, however, exploits the feature of "dimensions". Many algorithms, like the 3D transformation, operate on three or four dimensional data. For instance, color data has three dimensions (Red, Green, and Blue) and perhaps four (the Alpha channel). Geometry has four dimensions: X, Y, Z, and W.

Many of these "dimensional" algorithms have the feature that very similar sequences of operations are performed on all of the dimensions. For example, all four dimensions in Example 1 require dot products. Decreased latency exploits this feature by performing the similar sequences simultaneously with the SIMD hardware. The theoretical effect is that the latency of the algorithm is decreased by a factor equivalent to the number of dimensions on which operations are performed. However, the same algorithms usually exhibit some amount of dissimilarity. The dissimilarity introduces overhead which is not required in the scalar code, resulting in performance speedups which are typically somewhat lower than the theoretical maximum. In general, the decrease in performance is significantly less than that imposed by compressed computation. Therefore, it is recommended that decreased latency be used instead of compressed computation where possible. However, there is still a better, although less broadly usable, alternative.

2.2.3 Increased Throughput

Increased throughput is the simplest way to exploit parallelism. However, it does require that the algorithm is a loop which executes a significant number of iterations and that there are no dependencies between iterations. If those conditions are met, then increased throughput is implemented by executing four iterations of the loop at once. This requires replacing the existing scalar instructions with functionally equivalent SIMD instructions and modifying the loop counter. Notice that the length of each iteration has roughly the same number of instructions, but now only a quarter of the iterations are required. This method has the additional benefit that since there is no new overhead for the loop, the loop overhead is effectively reduced by 4x. This also can be called vectorization.

2.2.4 Maintenance Issues

Of the three types of exploitable parallelism, increased throughput offers the best potential for increasing code performance with Streaming SIMD Extensions. However, many data structures are arranged such that it is easiest to read in the data and immediately begin a compressed computation or decreased latency approach. This is a significant issue that currently deters many developers from implementing increased throughput algorithms. This issue, combined with the effort required to reorganize data structures such that they readily feed into an increased throughput algorithm, frequently causes developers not to consider increased throughput as a realistic alternative.

While compressed computation and decreased latency do appear to be solid alternatives in the short term, they do present some long term maintenance issues. Both methods require some amount of knowledge regarding the specific implementation of the hardware; for instance, the width of the operation (4 words, 8 bytes, etc.) as well as some specific behaviors of instructions. Relying on this level of hardware specific knowledge often makes it difficult to modify the implementation for another width or data type, and can require significant code rework. An increased throughput algorithm, on the other hand, requires no assumptions based on operation width or data type. These issues only affect the particular flavor (byte, word, float, etc.) of an instruction and the number of iterations that need to be performed. This helps to significantly reduce and simplify any coding rework or maintenance that may be required. For these reasons, only the scalar and increased throughput versions of the algorithm are discussed in this application note.

2.3 Development Environment Alternatives

The tools provided by Intel for developing code using the Streaming SIMD Extensions provide several different levels of abstraction for coding. The three that are discussed in this paper are inline assembly, intrinsics, and C++ classes, all of which currently require the Intel® C/C++ compiler. The three methods are covered from the highest abstraction layer to the lowest.

2.3.1 C++ Classes

The release includes a collection of files that make up a “Vector Class Library”. The intent of the library is to abstract the SIMD capabilities of both the MMX technology and the Streaming SIMD Extensions. Example 2 shows a piece of code that utilizes the new classes. Note that there is essentially no difference between this implementation and that shown in Example 1. The primary difference is that the “float” data type has been replaced with the new “F32vec4” type. Refer to Section 7 for the actual code and data structures used in this study.

The F32vec4 data type is important in that it serves two functions. First, it is typedef’ed based on an actual class in the library. The library is then responsible for generating the appropriate code using Streaming SIMD Extensions. The libraries also offer significant branch removal and elimination capabilities. That is, capabilities that can remove branches with either specialized instructions or conditional select techniques.

The second important function of F32vec4 is in the abstraction of the class names which denote the width of each class’s storage (*i.e*. 4 floats, 8 chars, etc.). By removing any width connotation from the type used in the code, it is possible to switch between whatever widths are supported (at this time either scalar floats or four, packed floats are supported). Modifying a single typedef from float to F32vec and changing the loop counter can switch this code from x87 to code that uses Streaming SIMD Extensions.

It is worth investigating the data structure used for this test case to understand why this switch is so easy. The data structure, which is located in the appendix, is referred to as a Structure-of-Arrays and is

essentially a “transpose” of the traditional Array-of-Structures used to store geometry data. The Structure-of-Arrays technique is usually the most convenient when working with SIMD. However, under certain circumstances an Array-of-Arrays is actually optimal. The Array-of-Arrays is equivalent to the Structure-of-Arrays but with the arrays running end-to-end instead of having individual pointers.

The last, and probably most significant, feature of this implementation is the ease of implementation and maintenance. The return on investment is staggering when compared to the other methods of implementing the algorithm using Streaming SIMD Extensions. In other words, note in Example 2 how closely the code resembles the original implementation in C.

```
void Transform_SIMD_c(VecInVerts &qInputVerts, VecOutVerts &qOutputVerts, int Counter)
{
    Fvec32 curX, curY, curZ, curW;
    Fvec32 mat00=1.0, mat01=0.0, mat02=0.0, mat03=0.0;
    Fvec32 mat10=0.0, mat11=1.0, mat12=0.0, mat13=0.0;
    Fvec32 mat20=0.0, mat21=0.0, mat22=1.0, mat23=0.0;
    Fvec32 mat30=0.0, mat31=0.0, mat32=0.0, mat33=1.0;
    Fvec32 VX_X=1.0, VX_Y=1.0, VX_Z=1.0;
    Fvec32 Xout, Yout, Zout;
    int i;

    for(i=0;i<Counter;i++)
    {
        curX    = qInputVerts.x[i];
        curY    = qInputVerts.y[i];
        curZ    = qInputVerts.z[i];

        curW    = rcp_nr((curX * mat30) + (curY * mat31) + (curZ * mat32) + mat33);

        Xout    =((((curX * mat00) + (curY * mat01) + curZ * mat02) +) * curW) + VX_X);
        Yout    =((((curX * mat10) + (curY * mat11) + curZ * mat12) +) * curW) + VX_Y);
        Zout    =((((curX * mat20) + (curY * mat21) + curZ * mat22) +) * curW) + VX_Z);

        qOutputVerts.x[i]= Xout;
        qOutputVerts.y[i]= Yout;
        qOutputVerts.z[i]= Zout;
        qOutputVerts.w[i]= curW;
    }
}
```

Example 2: C++ Class Implementation

2.3.2 Intrinsics Implementation

The primary reason for hand coding assembly is the ability to select the exact series of instructions. Sometimes it is also possible to do a superb job of scheduling the instructions in hand-coded assembly. However, the only area in which a human should be able to outperform a compiler is in the selection of the best instructions (because of a human’s ability to truly comprehend what operations are desired). Once the task of selecting the instructions is addressed though, a compiler should be able to select an optimal register allocation scheme and schedule. This is the fundamental principle behind intrinsics.

Example 3 shows an implementation of the test algorithm utilizing intrinsics for the Streaming SIMD Extensions. In addition to the ability to be scheduled and register allocated, intrinsics also have the benefit of not creating more basic blocks, a fundamental compiler concept. Basic blocks are pieces of code which are basically self contained. Very little optimization can occur across basic block boundaries, especially for instruction scheduling. Inlined-assembly (asm) blocks necessarily create such boundaries; therefore, they can limit optimizations that the compiler may be able to perform if the assembly block were not there. One obvious solution is simply to code the entire naturally occurring basic block in assembly. However, that increases development effort and decreases maintainability.

One important feature to notice about the code in Example 3 is that the data type has again been changed. The `__m128` data type is a fundamental data type currently unique to the Intel C/C++ compiler. It would be easy enough to abstract the `__m128` data type behind the F32vec4 data type as in Example 2; however, the intrinsics only operate on specific types of data, of which floats are not a part.

```
void Transform_SIMD_intrin(m128InVerts &m128InputVerts, m128OutVerts &m128OutputVerts, int VecCounter)
{
    __m128 curX, curY, curZ, curW, curW1, curW2, curW3, curW4;
    __m128 mat00=_mm_set_ps1(1.0), mat01=_mm_set_ps1(0.0), mat02=_mm_set_ps1(0.0),mat03=_mm_set_ps1(0.0);
    __m128 mat10=_mm_set_ps1(0.0), mat11=_mm_set_ps1(1.0), mat12=_mm_set_ps1(0.0),mat13=_mm_set_ps1(0.0);
    __m128 mat20=_mm_set_ps1(0.0), mat21=_mm_set_ps1(0.0), mat22=_mm_set_ps1(1.0),mat23=_mm_set_ps1(0.0);
    __m128 mat30=_mm_set_ps1(0.0), mat31=_mm_set_ps1(0.0), mat32=_mm_set_ps1(0.0),mat33=_mm_set_ps1(1.0);
    __m128 VX_X=_mm_set_ps1(1.0), VX_Y=_mm_set_ps1(1.0), VX_Z=_mm_set_ps1(1.0);
    __m128 Xout, Yout, Zout;
    int i=0;

    for(i=0;i<VecCounter;i++)
    {
        curX    = m128InputVerts.x[i];
        curY    = m128InputVerts.y[i];
        curZ    = m128InputVerts.z[i];

        curW1   =_mm_add_ps(_mm_add_ps(_mm_mul_ps(curX,mat30), _mm_mul_ps(curY,mat31)), _mm_add_ps(
                        _mm_mul_ps(curZ,mat32),mat33));

    //refine 1/W
        curW2   =   _mm_rcp_ps(curW1);
        curW3   =   _mm_add_ps(curW2, curW2);
        curW4   =   _mm_mul_ps(curW2, _mm_mul_ps(curW1, curW2));
        curW    =   _mm_sub_ps(curW3, curW4);
    //done refining

        Xout    =(_mm_add_ps(_mm_mul_ps(_mm_add_ps(_mm_add_ps(_mm_mul_ps(curX,mat00),
                _mm_mul_ps(curY,mat01)), _mm_add_ps(_mm_mul_ps(curZ,mat02), mat03)),curW),VX_X));
        Yout    =(_mm_add_ps(_mm_mul_ps(mm_add_ps(_mm_add_ps(_mm_mul_ps(curX,mat10),
                _mm_mul_ps(curY,mat11)), _mm_add_ps(_mm_mul_ps(curZ,mat12), mat13)), curW),VX_Y));
        Zout    =(_mm_add_ps(_mm_mul_ps(_mm_add_ps(_mm_add_ps(_mm_mul_ps(curX,mat20),
                _mm_mul_ps(curY,mat21)),_mm_add_ps(_mm_mul_ps(curZ,mat22), mat23)),curW),VX_Z));

        m128OutputVerts.x[i]=   Xout;
        m128OutputVerts.y[i]=   Yout;
        m128OutputVerts.z[i]=   Zout;
        m128OutputVerts.w[i]=   curW;
    }
}
```

Example 3: Intrinsics Implementation

Another obvious issue is the readability of an intrinsics implementation. The two alternatives are a tangled mess of what appear to be nested function calls, as shown in Example 3, or an endless listing of single function calls written to temporary local variables. While the readability and maintainability of intrinsics code may not be utopia, there may be a significant benefit to it. If a compiler is aware of scheduling tradeoffs between various processors, it should be possible to compile the code for whichever machines are supported by the compiler by simply modifying some compiler or pre-processor flags. This sort of compiler support is a definite win over re-coding the algorithm by hand for each machine.

An important fact to note is that the C++ classes used for Example 2 are based on the intrinsics. Since the classes are based on the intrinsics, it is reasonable to assume that the performance between the two implementations should be quite similar when the production release of the Intel C/C++ compiler becomes available.

2.3.3 Hand Assembly Implementations

Hand assembly coding always has been considered the pinnacle of performance, and one of the most critical aspects has been the scheduling of instructions. This was true in the era of the Pentium® processor architecture. However, Streaming SIMD Extensions will be introduced on a dynamic execution architecture, which is considerably less well understood. When implementing and optimizing an algorithm, it is important to understand the return on investment. To begin with, readability and maintainability typically decrease, resulting in significant costs at debug and support time, which is exactly when it can be least afforded.

The listings of the assembly code can be seen in the Transform.cpp source file in the samples/Transform directory. A dependency graph of the algorithm is shown in Figure 1.

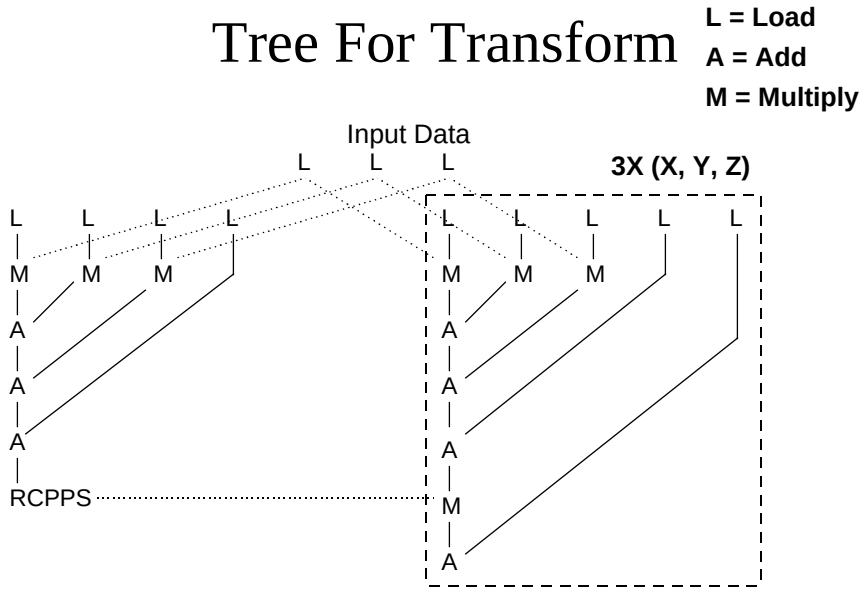

Figure 1: Dependency Graph for 3D Transform Algorithm

It has been found that dependency graphs can be uniquely helpful in scheduling code for Intel's dynamic execution processors. While there is a significant amount of information that can be gleaned from the graph, the most interesting aspect is the relationship between the graph and the schedule of the code in the Transform_SIMDasm_us1 function. Note that the code is separated into blocks of roughly four instructions each. The blocks were constructed by selecting operations from the graph which were as dissociated as possible. This was achieved by selecting the similar operation from each of the four major sub-graphs. Note that increased throughput is being used for SIMD, while the parallelism that could be exploited by decreased latency is being used to present as much instruction level parallelism in a given window of the instruction stream as possible.

The Transform_SIMDasm_us2 function represents a small amount of scheduling effort in addition to the _us1 version. The Transform_SIMDasm function, on the other hand, represents a significant amount of effort using a great deal of micro-architectural knowledge to develop the best schedule possible. The return on investment (effort required versus performance gain) with the compiler performance for the intrinsics and vector classes continually improving, puts a new perspective on the benefits of hand coding. The strategy for the initial version is important to note, because where you start from makes a big difference in how far you can go. A lot is also said for the abilities of the dynamic execution engine when it is given highly parallel code.

3 Conclusion

Three different implementation approaches have been presented along with some of their benefits and liabilities. While hand assembly coding can still achieve what is likely to be the highest performance possible, some of the new coding methods introduced with the Streaming SIMD Extensions are likely to become more attractive alternatives.

Source code examples can be found in: samples\Lighting and samples\Transform.cpp